Don't Loop, Iterate

Jonathan Amsterdam

Abstract: I describe an iteration macro for Common Lisp that is clear, efficient, extensible, and in excellent taste.
1 Introduction

Above all the wonders of Lisp's pantheon stand its metalinguistic tools; by their grace have Lisp's acolytes been liberated from the rigid asceticism of lesser faiths. Thanks to Macro and kin, the jolly, complacent Lisp hacker can gaze through a fragrant cloud of setfs and defstructs at the emaciated unfortunates below, scraping out their meager code in inflexible notation, and sneer superciliously. It's a good feeling.

But all's not joy in Consville. For—I beg your pardon, but—there really is no good way to iterate in Lisp. Now, some are happy to map their way about, whether for real with mapcar and friends, or with the make-believe of Series; others are so satisfied with do it's a wonder they're not C hackers.\(^1\) Still others have gotten by with loop, but are getting tired of looking up the syntax in the manual over and over again. And in the elegant schemes of some, only tail recursion and lambdas figure. But that still leaves a sizeable majority of folk—well, me, at least—who would simply like to iterate, thank you, but in a way that provides nice abstractions, is extensible, and looks like honest-to-God Lisp.

In what follows I describe a macro package, called iterate, that provides the power and convenient abstractions of loop but in a more syntactically palatable way. iterate also has many features that loop lacks, like generators and better support for nested loops. iterate generates inline code, so it's more efficient than using the higher-order function approach. And iterate is also extensible—it's easy to add new clauses to its vocabulary in order to express new patterns of iteration in a convenient way.

iterate is fully documented in AI Lab Memo No. 1236, *The Iterate Manual*.

2 More about Iterate

A Common Lisp programmer who wonders what's lacking with present-day iteration features would do well to consider setf. Of course, setf doesn't iterate, but it has some other nice properties. It's easy to use, for one thing. It's extensible—you can define new setf methods very easily, so that setf will work with new forms. setf is also efficient, turning into code that's as good as anyone could write by hand. Arguably, setf provides a nice abstraction: it allows you to view value-returning forms, like (car ... ) or (get ...) as locations that can be stored into. Finally and most obviously, setf looks like Lisp; it's got a syntax right out of setq.

iterate attempts to provide all of these properties. Here is a simple use of iterate that returns all the elements of num-list that are greater than three:

\(^1\)Hey, don't get mad—I'll be much more polite later, when the real paper starts.
(iterate (for el in num-list)
   (when (> el 3)
      (collect el)))

An iterate form consists of the symbol iterate followed by some Lisp forms. Any legal Lisp form is allowed, as well as certain forms that iterate treats specially, called clauses. for...in and collect are the two clauses in the above example. An iterate clause can appear anywhere a Lisp form can appear; iterate walks its body, looking inside every form, processing iterate clauses when it finds them. It even expands macros, so you can write macros that contain iterate clauses. Almost all clauses use the syntax of function keyword-argument lists: alternating keywords and arguments. iterate keywords don’t require a preceding colon, but you can use one if you like.

iterate provides many convenient iteration abstractions, most of them familiar to loop users. Iteration-driving clauses (those beginning with for) can iterate over numbers, lists, arrays, hashtables, packages and files. There are clauses for collecting values into a list, summing and counting, maximizing, finding maximal elements, and various other things. Here are a few examples, for extra flavor.

To sum a list of numbers:

(iterate (for i in list)
   (sum i))

To find the length of the shortest element in a list:

(iterate (for el in list)
   (minimize (length el)))

To find the shortest element in a list:

(iterate (for el in list)
   (finding el minimizing (length el)))

To return t only if every other element of a list is odd:

(iterate (for els on list by #'cddr)
   (always (oddp (car els))))

To split an association list into two separate lists (this example uses iterate’s ability to do destructuring):

(iterate (for (key . item) in alist)
   (collect key into keys)
   (collect item into items)
   (finally (return (values keys items))))
3 Comparisons With Other Iteration Methods

As with any aspect of coding, how to iterate is a matter of taste. I do not wish to dictate taste or even to suggest that iterate is a “better” way to iterate than other methods. I would, however, like to examine the options, and explain why I prefer iterate to its competitors.

3.1 do, dotimes and dolist

The do form has long been a Lisp iteration staple. It provides for binding of iteration variables, an end-test, and a body of arbitrary code. It can be a bit cumbersome for simple applications, but the most common special cases—iterating over the integers from zero and over the members of a list—appear more conveniently as dotimes and dolist.

do’s major problem is that it provides no abstraction. For example, collection is typically handled by binding a variable to nil, pushing elements onto the variable, and nreversing the result before returning it. Such a common iteration pattern should be easier to write. (It is, using mapcar—but see below.)

Another problem with do, for me at least, is that it’s hard to read. The crucial end-test is buried between the bindings and the body, marked off only by an extra set of parens (and some indentation). It is also unclear, until after a moment of recollection, whether the end-test has the sense of a “while” or an “until.”

Despite its flaws, do is superior to the iteration facilities of every other major programming language except CLU. Perhaps that is the reason many Lisp programmers don’t mind using it.

3.2 Tail Recursion

Tail-recursive implementations of loops, like those found in Scheme code [1], are parsimonious and illuminating. They have the advantage of looking like recursion, hence unifying the notation for two different types of processes. For example, if only tail-recursion is used, a loop that operates on list elements from front to back looks very much like a recursion that operates on them from back to front.

However, using tail-recursion exclusively can lead to cumbersome code and a proliferation of functions, especially when one would like to embed a loop inside a function. Tail-recursion also provides no abstraction for iteration; in Scheme, that is typically done with higher-order functions.

3.3 Higher-order Functions

Lisp’s age-old mapping functions, recently revamped for Common Lisp [3], are another favorite for iteration. They provide a pleasing abstraction, and it’s easy
to write new higher-order functions to express common iteration patterns. Common Lisp already comes with many such useful functions, for removing, searching, and performing reductions on lists. Another Common Lisp advantage is that these functions work on any sequence—vectors as well as lists.

One problem with higher-order functions is that they are inefficient, requiring multiple calls on their argument function. While the the built-ins, like map and mapcar, can be open-coded, that cannot be so easily done for user-written functions.

The second problem with higher-order functions is very much a matter of personal taste. While higher-order functions are theoretically elegant, they are often cumbersome to read and write. The unpleasant sharp-quote required by Common Lisp is particularly annoying here, and even in Scheme, I find the presence of a lambda with its argument list visually distracting.

Another problem is that it's difficult to express iteration of sequences of integers without creating such sequences explicitly as lists or arrays. One could resort to tail-recursion or do-times—but then it becomes very messy to express double iterations where one driver is over integers. Multiple iteration is easy in iterate, as illustrated by the following example, which creates an alist of list elements and their positions:

```lisp
(iterate (for el in list)
  (for i from 0)
  (collect (cons el i)))
```

### 3.4 Streams and Generators

For really heavy-duty iteration jobs, nothing less than a coroutine-like mechanism will do. Such mechanisms hide the state of the iteration behind a convenient abstraction. A generator is a procedure that returns the next element in the iteration each time it is called. A stream (in the terminology of [1]) is a data structure which represents the iteration, but which computes the next element only on demand. Generators and streams support a similar style of programming. Here, for example, is how you might enumerate the leaves of a tree (represented as a Lisp list with atoms at the leaves) using streams:

```lisp
(defun tree-leaves (tree)
  (if (atom tree)
      (stream-cons tree empty-stream)
      (stream-append (tree-leaves (car tree))
                    (tree-leaves (cdr tree)))))
```

Although tree-leaves looks like an ordinary recursion, it will only do enough work to find the first leaf before returning. The stream it returns can be accessed with stream-car, which will yield the (already computed) first leaf of the tree, or with stream-cdr, which will initiate computation of the next leaf.
Such a computation would be cumbersome to write using iterate, or any of the other standard iteration constructs; in fact, it is not even technically speaking an iteration, if we confine the term iteration to processes that take constant space and linear time. Streams, then, are definitely more powerful than standard iteration machinery.

Unfortunately, streams are very expensive, since they must somehow save the state of the computation. Generators are typically cheaper, but are less powerful and still require at least a function call. So these powerful tools should be used only when necessary, and that is not very often; most of the time, ordinary iteration suffices.

There is one aspect of generators that iterate can capture, and that is the ability to produce elements on demand. Say we wish to create an alist that pairs the non-null elements of a list with the positive integers. We saw above that it is easy to iterate over a list and a series of numbers simultaneously, but here we would like to do something a little different: we want to iterate over the list of elements, but only draw a number when we need one (namely, when a list element is non-null). The solution employ the iterate special clause next:

```
(iterate (for el in list)
  (for i from 1)
  (if el
    (collect (cons el (next i)))))
```

Using next with any driver variable changes how that driver works. Instead of supplying its values one at a time on each iteration, the driver computes a value only when a next clause is executed. This ability to obtain values on demand greatly increases iterate's power. Here, el is set to the next element of the list on each iteration, as usual; but i is set only when (next i) is executed.

3.5 Series

Richard C. Waters has developed a very elegant notation called Series which allows iteration to be expressed as sequence-mapping somewhat in the style of APL, but which compiles to efficient looping code [4].

My reasons for not using Series are, again, matters of taste. Like many elegant notations, Series can be somewhat cryptic. Understanding what a Series expression does can require some effort until one has mastered the idiom. And if you wish to share your code with others, they will have to learn Series as well. iterate suffers from this problem to some extent, but since the iteration metaphor it proposes is much more familiar to most programmers than that of Series, it is considerably easier to learn and read.
3.6 Prog and Go

Oh, don’t be silly.

3.7 Loop

Loop is the iteration construct most similar to iterate in appearance. Loop is a macro written originally for MacLisp and in widespread use [2]. It has been adopted as part of Common Lisp. Loop provides high-level iteration with abstractions for collecting, summing, maximizing and so on. Recall our first iterate example:

\[
\text{(iterate (for el in num-list) )}
\text{(when (> el 3))}
\text{(collect el))}
\]

Expressed with loop, it would read

\[
\text{(loop for el in list )}
\text{(when (> el 3)}
\text{(collect el))}
\]

The similarity between the two macros should immediately be apparent. Most of iterate's clauses were borrowed from loop. But compared to iterate, loop has a paucity of parens. Though touted as more readable than heavily-parenthesized code, loop's Pascalish syntax creates several problems. First, many dyed-in-the-wool Lisp hackers simply find it ugly. Second, it requires learning the syntax of a whole new sublanguage. Third, the absence of parens makes it hard to parse, both by machine and, more importantly, by human. Fourth, one often has to consult the manual to recall lesser-used aspects of the strange syntax. Fifth, there is no good interface with the rest of Lisp, so loop clauses cannot appear inside Lisp forms and macros cannot expand to pieces of loop. And Sixth, pretty-printers and indenters that don't know about loop will invariably display it wrongly. This is particularly a problem with program-editor indenters. A reasonably clever indenter, like that of Gnu Emacs, can indent nearly any normal Lisp form correctly, and can be easily customized for most new forms. But it can't at present handle loop. The syntax of iterate was designed to keep parens to a minimum, but conform close enough to Lisp so as not to confuse code-display tools. Gnu Emacs indents iterate reasonably with no modifications.

Indenting is a mere annoyance; loop's lack of extensibility is a more serious problem. The original loop was completely extensible, but the Symbolics version only provides for the definition of new iteration-driving clauses, and the Common Lisp specification does not have any extension mechanism. But extensibility is a boon. Consider first the problem of adding the elements of a list together, which can be accomplished with iterate by
(iterate (for el in list)
  (sum el))

and in loop with

(loop for el in list
  sum el)

But now, say that you wished to compute the sum of the square roots of the
elements. You could of course write, in either loop or iterate,

(iterate (for el in list)
  (sum (sqrt el)))

But perhaps you find yourself writing such loops often enough to make it worth-
while to create a new abstraction. There is nothing you can do in loop, but in
iterate you could simply write a macro:

(defmacro (sum-of-sqrts expr &optional into-var)
  '\(sum (sqrt ,expr) into ,into-var\))

sum-of-sqrts is a perfectly ordinary Lisp macro. Since iterate expands all
macros and processes the results, (sum-of-sqrts el) will behave exactly as if we’d
written (sum (sqrt el)).

There’s also a way to define macros that use iterate’s clause syntax. It’s fully
documented in The Iterate Manual.

4 Implementation

A Common Lisp implementation of iterate has existed for well over a year. It
runs under Lucid for HP 300’s, Sun 3’s and SPARCstations, and on Symbolics Lisp
machines. See The Iterate Manual for details.

5 Conclusion

Iteration is a matter of taste. I find iterate more palatable than other iteration
constructs: it’s more readable, more efficient than most, provides nice abstractions,
and can be extended.

If you’re new to Lisp iteration, start with iterate—look before you loop. If
you’re already using loop and like the power that it offers, but have had enough of
its syntax and inflexibility, then my advice to you is, don’t loop—iterate.
Acknowledgements

Thanks to David Clemens for many helpful suggestions and for the egregious pun near the end. Conversations with Richard Waters prompted me to add many improvements to iterate. Alan Bawden, Sundar Narasimhan, and Jerry Roylance also provided useful comments. David Clemens and Oren Etzioni shared with me the joys of beta-testing.

References


